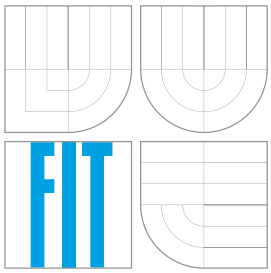
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYHLEDÁVÁNÍ VÝRAZŮ S NEDEFINOVANÝM CHOVÁNÍM V JAZYCE C

DETECTION OF EXPRESSIONS WITH UNDEFINED BEHAVIOR IN C LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ HELLEBRANDT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR MÜLLER

BRNO 2014

Zadání

Abstrakt

Práce se zabývá detekcí nedefinovaného chování v programech v jazyce C. Zaměřujeme se na nedefinované chování vznikající nesprávnou prací se sekvenčními body a vedlejšími efekty. Provedeme teoretický rozbor a pojmy jako nedefinované chování nebo vedlejší efekt zasadíme do kontextu práce. Dále vysvětlíme nebezpečnost konstrukcí vedoucích k nedefinovanému chování. Navrhneme metodu pro automatickou detekci zmíněného druhu nedefinovaného chování. Nakonec navrhneme a implementujeme nástroj pro jeho automatizovanou statickou detekci a popíšeme principy funkce tohoto nástroje. Při jeho návrhu klademe, narozdíl od současných řešení, důraz na funkčnost i v případě přístupu k paměti přes ukazatel nebo z volané funkce. Práce obsahuje příklady nebezpečných konstrukcí, na některých z nich jsou demonstrovány funkce vytvořeného nástroje.

Abstract

This thesis engages in detection of undefined behavior in the C language programs. We focus on undefined behavior stemming from incorrect work with sequence points and side effects. We perform a theoretical analysis and put terms like undefined behavior or side effect in context of the paper. Furthermore, we explain dangerousness of the constructs leading to undefined behavior. We propose a method for automated detection of the mentioned kind of undefined behavior. Finally, we design and implement a tool for its automated static detection and show the principles of the tool's function. While designing the tool, contrary to current solutions, we stress functionality even in cases like accessing the memory via a pointer or from a called function. The thesis contains examples of dangerous constructs, functions of the created tool are demonstrated on some of them.

Klíčová slova

Clang, LLVM, jazyk C, nedefinované chování, statická analýza, vedlejší efekt, sekvenční bod

Keywords

Clang, LLVM, C language, undefined behavior, static analysis, side effect, sequence point

Citace

Lukáš Hellebrandt: Detection of Expressions with Undefined Behavior in C Language, bakalářská práce, Brno, FIT VUT v Brně, 2014

Detection of Expressions with Undefined Behavior in C Language

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Petra Müllera.

.....
Lukáš Hellebrandt
May 19, 2014

Poděkování

Děkuji Ing. Petru Müllerovi za vedení mé práce a mnoho inspirujících podnětů.

Contents

Chapter 1

Introduction

The C programming language is a very powerful and efficient language for writing programs. Arguably the most important reason for this is that it is almost as low-level as a structured programming language can be. This leads to great freedom of choice and potentially very efficient resulting programs. With this freedom, however, comes a programmer's responsibility to avoid some language constructs to prevent possible unpredictable results – the undefined behavior in C.

Our goal is to describe possible sources of undefined behavior related to sequence points and side effects (described in the rest of this chapter), design a tool for automated static detection of this behavior and ultimately implement it. We will describe the necessary theoretical background, mention current solutions of this problem and their flaws. We will argue about the best way of implementing a new tool without these flaws and show some problems that must be overcome in order to do it. We will also make a set of examples. The final implemented tool will be tested on these examples.

In this chapter, we will describe the necessary theoretical background. The most important document to study regarding the C language is its most recent standard: ISO/IEC 9899:2011 [13] , informally known as *ISO C11*. (For the purpose of this paper, we are going to use its last free and public draft available [20])

1.1 Undefined behavior

ISO C11 specifies how the compiler implementation is expected to work and when there are no expectations regarding the implementation's behavior at all. Multiple terms are used to describe similar but different requirements on the implementation: [20]

- *Behavior* – external appearance or action
- *Unspecified behavior* – use of an unspecified value, or other behavior where the C standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance
- *Implementation-defined behavior* – unspecified behavior where each implementation documents how the choice is made
- *Locale-specific behavior* – behavior that depends on local conventions of nationality, culture, and language that each implementation documents

- *Undefined behavior* – behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the C standard imposes no requirements

Bear in mind that undefined behavior allows *any* behavior. That means not only the returned value of the operation with undefined behavior or value of some variable is not specified but also that the implementation is allowed to do *anything*. The program might run "just fine" (as the programmer expected) or the program might erase your hard disk and it would be perfectly legal according to the standard.

1.2 Side effect

ISO C11 [20] gives the following definition of a side effect: "Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an L-value expression includes determining the identity of the designated object."

Informally said, a side effect is usually modifying a memory during evaluation of an expression. For example `i++` is an expression whose *return value* is the value of `i` before evaluation *and the value of* `i` *is incremented by 1* – this is the side effect. The expression evaluation accesed the memory addressed by `i` and modified its value.

1.3 Sequence point

The C standard [20] states that "Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is sequenced after A.) If A is not sequenced before or after B, then A and B are unsequenced. Evaluations A and B are indeterminately sequenced when A is sequenced either before or after B, but it is unspecified which. The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B."

Informally said, the evaluation order of expressions may be specified, but in many cases it is not. If it is not, the implementation is in certain cases required to specify it and perform the evaluation consistently and predictably. ISO C11, in other words, states that an expression A must be evaluated (or at least the external behavior must appear like it is evaluated – "an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced" [20]) before an expression B when it is on the left side of a sequence point that is between A and B. Otherwise A and B may either be evaluated in an unspecified order or they can even *not be evaluated in any order* – the evaluation of A may start, then be interrupted, B may be evaluated and A evaluation may be completed then, for example.

ISO C11 enumerates the sequence points as follows: [20]

- Between the evaluations of the function designator and actual arguments in a function call and the actual call.

- Between the evaluations of the first and second operands of the following operators: logical AND & ; logical OR | ; comma , .
- Between the evaluations of the first operand of the conditional ? : operator and whichever of the second and third operands is evaluated.
- The end of a full declarator
- Between the evaluation of a full expression and the next full expression to be evaluated. The following are full expressions: an initializer that is not part of a compound literal; the expression in an expression statement; the controlling expression of a selection statement (if or switch); the controlling expression of a while or do statement; each of the (optional) expressions of a for statement; the (optional) expression in a return statement.
- Immediately before a library function returns.
- After the actions associated with each formatted input/output function conversion specifier.
- Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call.

1.4 Consequences

According to ISO C11 [20], undefined behavior occurs when "a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object". That means whenever there are two expressions that are unsequenced and have multiple side effects *or* a side effect and an access on the same object (meaning its value is to be modified, but in any order and possibly not even in any order), the undefined behavior occurs. This means not only the value of a given object is undefined but the whole program might do absolutely anything – most of the implementations will probably do something "reasonable", but they are not required to.

This kind of undefined behavior appears not to be very dangerous on sight – most of the programmers would either expect, e.g., `i = i++;` expression to be well-defined or think the value of `i` would either be `i` or `i+1`. However, if you write `i = i++;` in your program and run it, you might get your hard disk erased just as well.

Another reason why this is dangerous is that most of the implementations will behave "reasonably" and do one of the things a programmer would expect. It is therefore hard to reveal this kind of errorneous C constructs. That might be present in a program for many years until the implementation behavior changes and the program, too, suddenly changes its behavior for no obvious reason.

Chapter 2

Examples

As described in the previous chapter, it is necessary to avoid having any undefined behavior in one's program. A program with undefined behavior is poorly written and needs to be fixed. Also it is highly questionable whether even a very good programmer can manage to find all of the possible constructs leading to undefined behavior due to sequence point misunderstanding as some of them are very obscure. Most of the people who can code in C have probably never even heard of "sequence point" and often do not believe certain construct leads to undefined behavior or are confused and try to guess which is the "correct" result.

2.1 Undefined behavior examples

In this section, we will show some examples of constructs leading to undefined behavior from most basic ones to some more obscure ones.

Example 1 is a code that we have shown some people to give an example of what our work is about. Many of them, however, did not even believe this is an undefined behavior, which only shows how dangerous even the most basic case is.

Figure 2.1: Example 1

```
1 int main(void){
    int i = 1;
3   i = i++;
    return i;
5 }
```

The assignment operator modifies the same memory as the post-increment operator while there is no sequence point between them. Not only does this operation have unspecified results (which most of the people who have seen it for the first time admitted after our explanation) but it leads to undefined behavior rendering the whole program meaningless and dangerous.

Example 2 is almost the same except for one thing: the second operand of the assignment is now accessed via pointer.

Figure 2.2: Example 2

```
int main(){
  int i = 0;
  int *j = &i;
  i = (*j)++;
    return i;
}
```

It is, however, the same undefined behavior as in the previous case because it modifies the same memory more than once within the same sequence point. It is also not so easily observable and one can miss this construct when not looking for it specifically.

Example 3 shows undefined behavior when calling a function:

Figure 2.3: Example 3

```
int f(int *a){
    (*a)++;
    return *a;
}

int g(int *b){
    (*b) *= 2;
    return *b;
}

int main(){
    int i = 1;
    i = f(&i) + g(&i);
    return i;
}
```

One problem here is that there is no defined order of evaluation of the addition's operands. Many people would think that the result is just unspecified and will be one of the two values. Another problem, however, is that both of these functions modify the same memory, so this is, again, undefined behavior.

Example 4 is another undefined behavior case which some of the good programmers may miss:

While the standard states that a *comma operator* is a sequence point, in this case, the comma does not denote an *operator*. The order of evaluation of the function arguments (calling `a` and `b` functions) is undefined and, in this particular case, it leads to undefined behavior as both of the functions have a side effect on the same variable.

Example 5 shows the same problem when indexing arrays:

This is an example of unsequenced modification *and* read during the same sequence point. Not only don't we know the memory location and the value written, the program has undefined behavior.

Figure 2.4: Example 4

```
int a(int *a){
    (*a)++;
    return *a;
}

int b(int *b){
    (*b) *= 2;
    return *b;
}

int f(int a, int b){
    return a+b;
}

int main(void){
    int i = 1;
    //This is NOT a comma OPERATOR
    i = f(a(&i), b(&i));
    return i;
}
```

Figure 2.5: Example 5

```
int main(void){
    char arr[42];
    int i = 0;
    arr[i++] = i;
    return arr[i-1];
}
```

Example 6 shows how usage of macro or other obfuscation can make the undefined behavior even harder to find:

Figure 2.6: Example 6

```
#define MACRO i = (*j)++;
int main(){
  int i = 0;
  int *j = &i;
  MACRO
  return i;
}
}
```

This is the same as Example 2, only with a simple macro used. It makes the undefined behavior harder to observe.

Example 7 demonstrates an obvious case of undefined behavior where a programmer who is not aware of sequence point problematic would not expect any problem as the result seems to be the same regardless of the evaluation order.

Figure 2.7: Example 7

```
int main(void){
    int j;
    int i = 0;
    j = i++ + ++i;
}
```

The difference between postfix and prefix incrementation is that, while they both increment the variable value, the postfix one returns the original value of the variable and the prefix one returns the incremented value. Many people would probably guess that no matter in which order the expression is evaluated, it will return the same value. Saying that would be the same mistake as we have shown in some of the previous examples as this expression modifies the same memory more than once within the same sequence point.

2.2 Positive examples

These are all examples of *undefined behavior* where the implementation is not required to do anything particular. This is due to unsequenced modifications (or modifications and access) of the same memory location within one sequence point. We will now show some "positive" examples of programs *without* any unspecified results or undefined behavior.

Positive example 1 is a code that, while it might look suspicious, is perfectly fine:

Figure 2.8: Positive example 1

```
int main(void){
    int i = 42;
    int j;
    j = i++ , ++i;
    return j;
}
```

This is correct because a *comma operator* is a sequence point.

Positive example 2 is another example of correct code.
The return value of `i++` is used in an assignment, but in another object.

Positive example 3 might look suspicious for a person over-concerned with the sequence point problem.
While on first sight this looks similar to `i = ++i;`, the `i+1` expression has no side effect on `i` and this code is correct.

Figure 2.9: Positive example 2

```
int main(void){
    int i = 42;
    int j, k;
    k = (j = i++);
    return k;
}
```

Figure 2.10: Positive example 3

```
int main(void){
    int i = 42;
    i = i + 1;
    return i;
}
```

Chapter 3

Existing solutions and the goal of this work

As we described in the previous chapter, sequence-point related undefined behavior is potentially very dangerous and hard to detect before the real problem occurs. Some compiler implementations have very basic warnings for a programmer at compilation time, but most of them fail in any non-trivial case (as demonstrated in 3.3), for example using a function with a side effect or accessing the object via a pointer rather than directly. We will therefore try to compare some of these implementations, describe why their solutions are not powerful enough and briefly describe our concept of an automated tool for detection of the previously described behavior.

3.1 Aliasing

The mentioned trivial case `i = i++;` is relatively easy to detect and many compilers are able to detect it. We can determine it leads certainly to undefined behavior. But this decision is much more complicated when we consider using pointers as in this simple case:

Figure 3.1: Aliasing problem example

```
1 int main(){
    int i = 0;
3   int *j = &i;
    i = (*j)++;
5     return i;
  }
```

As `i` points to the same memory location as `*j`, i.e. if the two variables *alias*, the example still leads to the undefined behavior. Otherwise, however, it would not, because there would not be more than one memory modification *to the same memory location*. Therefore we need to be able to determine whether the two variables alias in order to detect undefined behavior considering the use of pointers.

3.2 Static analysis

As said at the beginning, our goal is implementing a tool that is able to detect undefined behavior *statically*. Static program analysis is analysis of the program without actually executing the program [11]. In this case, it will be performed on the C code we want to test and possibly on the object code generated from it. In the following sections, we will focus mainly on static analysis solutions, but we will also mention some other possibilities.

3.3 Existing solutions

Currently we know about two existing tools capable of very basic detection of sequence point related undefined behavior that are included in widely used compilers – Clang [3] and GCC [8]. Another tools are PVS-Studio [10], Cppcheck [7], Astrée [1] and Coverity [6], that are specifically meant for static analysis. Errors caused by such undefined behavior occurrences can also be detected at runtime with dynamic analysis tools such as CompCert [5].

3.3.1 GCC

GCC's `-Wsequence-point` modifier is able to detect some basic occurrences of side-effect related undefined behavior, but fails in cases as basic as using a function with side effect or side effect on a memory accessed via pointer.

The example output of compilation of a Example 1, *without* pointers:

```
$ gcc --std=c11 -Wall test1.c
2
test1.c: In function 'main':
4 test1.c:3:5: warning: operation on 'i' may be undefined [-Wsequence-point]
  i = i++;
6   ^
```

Example 2 *with* pointers doesn't generate any warning. We used GCC version 4.8.2 for this test.

3.3.2 Clang

Clang is an alternative to GCC. Its compile-time warnings fail in the same basic cases as GCC. It is specifically designed to be modular, reusable, its source code and API are easily understandable [4]. Clang therefore offers a reasonable environment for further development and will be used for a part of our work.

The example output of compilation of an Example 1, *without* pointers:

```
$ clang --std=c11 -Wall test1.c
2 test1.c:3:8: warning: multiple unsequenced modifications to 'i' [-
    Wunsequenced]
  i = i++;
4   ~  ^
1 warning generated.
```

Example 2 *with* pointers doesn't generate any warning. We used Clang version 3.5.0 for this test.

3.3.3 PVS-Studio, Cppcheck, Astrée, Coverity

PVS studio, Cppcheck, Astrée and Coverity are static analyzers. They can find possible errors in source code of various programming languages, all of them including C. While PVS-Studio might be an interesting option, it has the same problems as previously mentioned ones. We didn't find a way to make Cppcheck find sequence-point realted undefined behavior at all. We didn't manage to test Coverity and Astrée. Also, because these programs are meant specifically for static checking, it would be hard to reuse a check in a compiler so that warnings show at compile-time.

3.3.4 CompCert C

CompCert is a formally verified compiler [5]. It is able to modify the resulting binary so that *dynamic* check is performed. Since this happens *at run-time* and our goal is to design a tool able to detect undefined behavior *statically*, we can not use it.

3.4 Theoretical work

Multiple theoretical works related to this problem exist, none of them, however, covers all the needed topics:

- A blogpost [14] about the extension to CompCert. Its goal is detection of the same kind of errors as we discuss in this paper. The problem of pointer use is also mentioned in this blogpost. The extension is based on formal semantics of C by Chucky Ellison and Grigore Rosu. [12]. It performs the analysis at run-time.
- A paper published by WG14 about dividing the AST into subexpressions with sequence points specifically in mind. [19]
- The original inspiration for this thesis. It shows some examples of undefined behavior and states that it would be useful if there were some dynamic checkers. [16]
- One of multiple papers about alias analysis. [18] It could be used to solve the problem with pointer usage. However, we will not need to implement our own alias analysis as shown later.

3.5 Sequence-point analyzer goal

Both Clang and GCC are able to statically detect undefined behavior in Example 1 (`i = i++;`). None of them, however, works e.g. on Example 2 or Example 3 (`i = (*j)++;` and `i = f(&i) + g(&i);`). This is due to lack of alias analysis and not considering functions at all.

These solutions are not sufficient as they usually only catch undefined behavior in example and test source codes, which is so apparent that most programmers would not do such a mistake. Most of these errors occur while dealing with functions that have side effects, accessing a variable via a pointer, using array indexation etc. There are multiple theoretical works regarding this topic, none of them, however, deals satisfyingly with pointers (where alias analysis is needed) [15] and functions.

As there is no suitable existing tool, our work's goal is to *implement a static checker for side-effect and sequence-point related undefined behavior, including access via pointers and functions.*

Chapter 4

Sequence-point analyzer

In the most basic case, ignoring functions and pointers, this tool's goal would be trivial: depending on the type of the statement and on side effects of its operands, output a set of tuples $(position, function, variable)$ where the behavior is undefined. This would be easily possible given access to the abstract syntax tree (AST) of the program, which is provided by Clang.

However, as shown before, aliasing poses a major problem and needs to be kept in mind while developing a concept of the whole tool. As pointers are widely used in C, we need to be able to determine whether two variables point to the same memory location or not.

Clang itself doesn't have a tool for this and doesn't allow us to easily bind some external alias analysis to the AST at compile-time. If we wanted to perform the alias analysis on the AST level in Clang, we would need to implement our own alias analysis. It is possible and there are multiple ways to do this. It is, however, neither necessary nor is it our goal. There are multiple implementations of alias analysis that can be used. The remaining problem is that we can not perform alias analysis during the compilation and therefore need to perform it completely out of the Clang part.

4.1 High-level architecture

For reasons shown in the previous chapter, the SPA will consist of two phases. The first phase is generating a set of constraints under which the behavior is undefined. This part is implemented as a plugin for the Clang compiler. The second phase will check these constraints and produce the final output. This phase is implemented as a Bash script using alias analysis implemented in LLVM.

Provided we have access to the AST, we can detect some *potential* occurrences of undefined behavior. The AST gives us information on variables and operations they are involved in. However, without implementing our own alias analysis, we are not able to determine whether two variables alias or not – hence "potential" undefined behavior. This said, the goal of the first part of our project is to merely generate a set of constraints saying under what conditions the undefined behavior may occur. For our Example 2, the goal is to generate a rule saying „On the row 4, there is undefined behavior *if* `*j` aliases with `i`". The exact method of generating these constraints will be shown later. We have necessary access to the AST from the Clang plugin. A Clang plugin is also an option that fully uses Clang's modularity. It can be run at compile-time, can be ommited from running, and is extendable easily as we have access to most of the information Clang can provide.

Figure 4.1: The problem with functions

```
int f(int *a){
    (*a)++;
    return *a;
}

int g(int *b){
    (*b) *= 2;
    return *b;
}

int main(){
    int i = 1;
    i = f(&i) + g(&i);
    return i;
}
```

This set of constraints can be checked later on either in LLVM or using some external alias analysis, and a script for that can be considered the second part. This constraint-checking part will be implemented as a bash script, that uses alias analysis in LLVM. The alias analysis implemented in LLVM can be run on the LLVM IR ("intermediate representation", "bitcode"). The reason we can not do this directly is that undefined behavior in the C source code is already well-defined after translating to the LLVM IR. We can not find any undefined behavior in LLVM IR as it does not need to correspond to the original C source code at all (and indeed, it does not as LLVM IR is a intermediate representation suitable from multiple programming languages). This means we need to map some parts of the LLVM IR, which the alias analysis will be run on, to the original source code. We will use the debugging information for this purpose.

The following two sections will describe the exact methods of achieving each part's goals. They will be run one after another, the Constraint Generating part first and the Constraint Checking part second using the first part's output.

4.1.1 Functions

We are not aware of any alias analysis capable of determining whether two objects in different functions alias.

In the case shown in the figure 4.1, the most performance-efficient and easiest approach is just stating that if the function *can* use its argument as an L-value (meaning it has the object pointer or some object it can get it from as an argument), we will assume it *does*. This can be improved later either by using some implementation of cross-function alias analysis or changing the implementation so we change the output format to allow logical operations between constraints and checking which function arguments have a side effect on them in the function. Whether these parameters *do* have a side effect on them in that function would then be logically multiplicated with the constraint the function is called from.

The second option would not be easily reusable and would do a little or no improvement to the tool as it is a prototype showing the methods described in this paper. We believe this issue should be solved on the alias analysis side and the tool's goal should be merely

Figure 4.2: Example 1 AST dump

```
TranslationUnitDecl 0x29bdec0 <<invalid sloc>>
|-TypedefDecl 0x29be3c0 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x29be420 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x29be770 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]
    '
`-FunctionDecl 0x29be880 </examples/test1.c:1:1, line:5:1> main 'int (void)'
  `-CompoundStmt 0x29beb18 <line:1:15, line:5:1>
    |-DeclStmt 0x29bea08 <line:2:2, col:11>
    | `-VarDecl 0x29be990 <col:2, col:10> i 'int'
    |   `-IntegerLiteral 0x29be9e8 <col:10> 'int' 1
    |-BinaryOperator 0x29bea90 <line:3:2, col:7> 'int' '='
    | |-DeclRefExpr 0x29bea20 <col:2> 'int' lvalue Var 0x29be990 'i' 'int'
    | `-UnaryOperator 0x29bea70 <col:6, col:7> 'int' postfix '++'
    |   `-DeclRefExpr 0x29bea48 <col:6> 'int' lvalue Var 0x29be990 'i' 'int'
    `-ReturnStmt 0x29beaf8 <line:4:2, col:9>
      `-ImplicitCastExpr 0x29beae0 <col:9> 'int' <LValueToRValue>
        `-DeclRefExpr 0x29beab8 <col:9> 'int' lvalue Var 0x29be990 'i' 'int'
```

generating constraints which can then be checked by some future implementation of cross-function alias analysis.

4.1.2 Restrict keyword

The `restrict` keyword means that the pointer variable declared as `restrict` *and* pointers derived from it are the only pointers of that type used pointing to the memory that variable points to. [20] This applies to the whole lifetime of that variable.

That means declaring two variables as `restrict` in the same scope means they do not alias with each other. There will be no constraints generated where two different variables are declared as `restrict`. However, `restrict` variables can not be omitted completely because there can still be side effects implied on them and those side effects do not need to be sequenced.

4.2 Generating constraints

A Clang plugin has access to the program's AST. Given this AST, we can find statements (nodes of the AST), where the undefined behavior can (but might not, depending on the results of the alias analysis) occur.

4.2.1 Division into sequence points

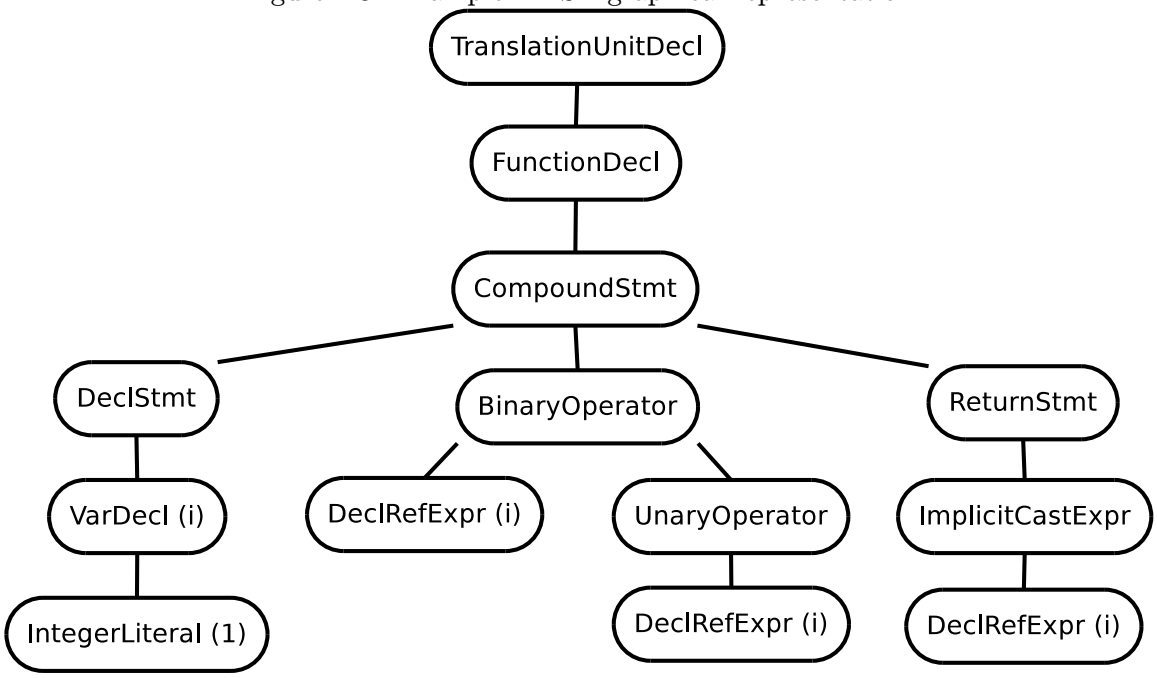
AST dump of Example 1 provided by Clang is shown on figure 4.2.

On figure 4.3 is its graphical representation (after omitting the typedefs from the beginning).

As we can see, we can determine what (if any) child node is the side effect going to be applied to, by definition of individual AST nodes.

When a child node has a side effect on certain variable, the parent also has side effect on that same variable. Let there be a node A and a node B that is A's ancestor. When we

Figure 4.3: Example 1 AST graphical representation



go several levels up from node A towards the root and stop in node B, we can see the side effect can not disappear. The subtree of B in which the node A is still has the side effect on the same memory address.

There is a special type of node in the AST – a reference to variable declaration. It is called DeclRefExpr in Clang and is always a leaf of the AST. We therefore can start from the DeclRefExpr nodes and while traversing the tree up to the root, decide whether the current node implies a side effect on a given DeclRefExpr.

4.2.2 L-value scope

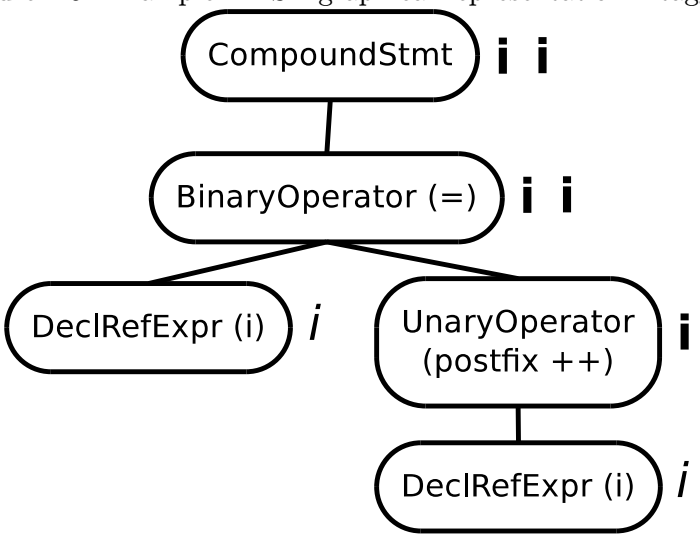
Some expressions keep an address of the given operand, meaning it can be used later in some parent node as an L-value and, e.g., be assigned to. Consider the code in Figure 4.4.

Figure 4.4: Example of limited L-value scope

```
int main(void){
    int i = 2;
    int a = 3;
    *(a==3?&i:&i) = 42;
    return i;
}
```

In this example, it is the case of `&i`. Some other expressions, however, get rid of the operand address irreversibly and we can be sure it will never be used as an L-value until its next occurrence. An example of this is the `a` variable in the conditional operator from the code above: it can never more be used as an L-value in the whole subtree with the assignment it appears in as the conditional operator just uses it and throws it away.

Figure 4.5: Example 1 AST graphical representation – tagged



4.2.3 Node tagging

From this, we can conclude that every node can be tagged either as „implying a side effect“, „not implying anything“ or „making side effect impossible“ on certain memory place given by certain DeclRefExpr. We can do this for each DeclRefExpr. The picture from the above would then be tagged as shown on figure 4.5.

We began tagging from the left DeclRefExpr. It surely can be used as an L-value but the expression itself implies no side effect, hence the tag in italic („not implying anything“). We went towards the root and as we passed the `BinaryExpression (=)`, we mark it as an expression implying a side effect on `i` (tag in bold) as it modifies the memory contents of `i`. We continued to the top implicitly marking all the parent nodes as having a side effect on `i`. Then we can do the same for the right DeclRefExpr and as we see two tags in bold of the same variable, we can say we found an undefined behavior.

This method can also be described by the following algorithm:

```
 1: for declRefExpr in declRefExprs do
 2:     node = declRefExpr
 3:     while node.type != functionDecl do
 4:         if nodeImpliesSideEffect(node) then
 5:             tagNodeAndAllPredecessors(node, declRefExpr)
 6:             break
 7:         else if nodePreservesLvalue(node) then
 8:             node = node.parent
 9:         else
10:             break
```

4.2.4 Output of the Constraint Generating part

For sake of portability, the output of the first phase is a plaintext directed to the standard output of the compiler. The output must contain the following information:

- Function in which the variables are to be checked for aliasing

Figure 4.6: Output of the first part of the SPA

```
main 16 9 16 19 *j *k
main 16 9 16 19 *k *j
main 16 5 16 19 *j i
main 16 5 16 19 *k i
```

- Location of the statement with potentially undefined behavior
- Variables to be checked; if they alias, undefined behavior occurs

Each possible undefined behavior is on one line. The format is
"`function b_row b_col e_row e_col variable1 variable2`" if the constraint needs to be checked because `variable1` and `variable2` are different, or
"`function b_row b_col e_row e_col variable`" if no further check is needed as the possible undefined behavior comes from occurrences of the same variable. `b_row` and `b_col` denote the row and column of the beginning of statement the undefined behavior occurs in and `e_row` and `e_col` denote its end.

This output can then be used by any external alias analysis.

4.3 Checking the constraints

Once the constraints are generated, there are many means of checking them. Any alias analysis may be used and it is the alias analysis used what decides an algorithm used for comparing it with the constraints as the output and information received from the alias analysis may vary for each implementation. We will further describe our actual implementation which is a bash script using a built-in alias analysis of the LLVM. We will use the Eexample 2 (`i = (*1)++`) for the following examples.

As we said previously, it is necessary to map the LLVM IR to the original source code. This requires two steps – translating temporary variable names to those of the original variables and mapping to the original location in the source code. Then, the constraints generated in the previous part can be compared to these translated aliases.

4.3.1 Alias analysis implementation

We used the alias analysis implemented in the LLVM. It offers multiple versions of alias analysis and after testing, the "basic" alias analysis was evaluated as best. This implementation of alias analysis is simple and leads to many indefinite answers but in our case, it led to at least some results while other implementations seemed absolutely useless. Considering our approach to indefinite answers mentioned later and the results of the tests, the "basic" version seems to be absolutely sufficient.

Other implementations' failures were probably caused by the fact that they depend heavily on compiler optimizations [17] which, however, need to be turned off completely. Searching for aliases in the optimized code and trying to match them to the original source would make no sense. Testing shows that in many cases, the alias analysis would just stop with a "no pointers" error meaning anything that could be checked has been optimized out.

Figure 4.7: Output of the alias analysis

```
$ opt -disable-output -basicaa --aa-eval -print-all-alias-modref-info test1A.
    bc
Function: main: 4 pointers, 2 call sites
  NoAlias:   i32* %i, i32* %retval
  NoAlias:   i32* %retval, i32** %j
  NoAlias:   i32* %i, i32** %j
  NoAlias:   i32* %0, i32* %retval
  MayAlias:  i32* %0, i32* %i
  NoAlias:   i32* %0, i32** %j
```

Figure 4.8: LLVM IR

```
$ llvm-dis test1A.bc -o -
define i32 @main() #0 {
  call void @llvm.dbg.declare(metadata !{i32* %i}, metadata !12), !dbg !13
  store i32 0, i32* %i, align 4, !dbg !14
  call void @llvm.dbg.declare(metadata !{i32** %j}, metadata !15), !dbg !17
  store i32* %i, i32** %j, align 8, !dbg !18
  %0 = load i32** %j, align 8, !dbg !19
  %1 = load i32* %0, align 4, !dbg !19
  %inc = add nsw i32 %1, 1, !dbg !19
  store i32 %inc, i32* %0, align 4, !dbg !19
  store i32 %1, i32* %i, align 4, !dbg !19
  %2 = load i32* %i, align 4, !dbg !20
  ret i32 %2, !dbg !20
```

4.3.2 Output of the alias analysis

Figure 4.7 shows part of the output of alias analysis of the Example 2. We can easily filter out the non-aliasing variables. Out of possible answers – NoAlias, MayAlias, MustAlias – the latter two are considered to confirm that an undefined behavior occurs. This may lead to false positives depending on alias analysis used – MayAlias is not a definite answer and we simply choose possible false positives over false negatives.

We see there are many temporary variables (the numbered ones) and not the original names of the variables. In order to match these temporary variables to real variable names, we need the LLVM IR of the original program.

The alias analysis can be run on the LLVM IR ("intermediate representation", "bitcode"), which can also be printed as a text and also can contain debugging information. These three elements are necessary for the following approach to work.

4.3.3 Alias translation

On figure 4.8, there is a part of the LLVM IR necessary for translation of the temporary variables to their original names. On that example, we can see that `\%0` matches `*j` in the original source file.

Furthermore, we can see a link to debuginfo `!dbg !19` attached which can be used in the next part.

For programs with multiple functions, the results are similar to those demonstrated and LLVM IR keeps the division into functions so we are able to filter out all the functions we do not currently want to match against. This is important because temporary variable names are not unique across functions.

4.3.4 Matching translated aliases to their original location

Figure 4.9 shows the debugging information necessary to match the aliasing info to the original source code. Using the link to debugging information from a LLVM IR, we can find the correct debugging information which contains information on location in the source file and the function name. The row is the first argument, the column the second argument and the third argument is a link to another debugging information (in this case `!dbg !4`) which contains information about the function the undefined behavior occurs in.

In order to get access to this debugging information, we need to compile to the bitcode using `-g3` (for debugging information in general) and `-gcolumn-info` (for columnt information, otherwise it would be always 0 for performance reasons) parameters.

4.3.5 Constraints and translated aliases comparison

The described method will lead to a set of constraints from the first part of the SPA (figure 4.6) *and* a set of aliases matched with their original names and locations ("translated aliases"). It is then a matter of comparing two lists, each item with each, and if some of them matches, it indicates the undefined behavior.

As we found no way of getting the exact position of the *statement* where two variables alias out of the LLVM alias analysis, we need to introduce a solution possibly leading to false positives: If two variables have aliased *anywhere before the statement, and in the same function as the statement occurs*, it is considered to alias in that statement, too. This is a flaw we were not able to circumvent. Example of this case is in figure 4.10, where `*k` is considered as aliasing with `i` and undefined behavior is falsely detected. This is a flaw of the second part of the SPA and depends on information accessible from the alias analysis. Alias analysis errors and indefinite answers may also lead to false positives.

4.3.6 Output format

The output of the second part is meant to be the final output of the SPA for the user. It shows actual undefined behavior occurrences (in contrast with the first part showing all *possible* occurrences), the function name, the *approximate position* and names of the variable(s) causing the undefined behavior.

Example output:

```
1 Possible undefined behavior in function "main" at [16,9] - "*j" aliases with
    "*k"
  Possible undefined behavior in function "main" at [16,5] - "*j" aliases with
    "i"
3 Possible undefined behavior in function "main" at [16,5] - "*k" aliases with
    "i"
```

Figure 4.9: Debug info

```
$ llvm-dis test1A.bc -o -
!0 = metadata !{i32 786449, metadata !1, i32 12, metadata !"clang version 3.5.0 (trunk 205669)", i1 false, metadata !"", i32 0, metadata !2, metadata !2, metadata !3, metadata !2, metadata !2, metadata !"", i32 1} ; [ DW_TAG_compile_unit ] [/home/lhellebr/bc/new/SPA/SPA/examples/test1A.c] [DW_LANG_C99]
!1 = metadata !{metadata !"SPA/examples/test1A.c", metadata !"/home/lhellebr/bc/new/SPA"}
!2 = metadata !{}
!3 = metadata !{metadata !4}
!4 = metadata !{i32 786478, metadata !1, metadata !5, metadata !"main", metadata !"main", metadata !"", i32 3, metadata !6, i1 false, i1 true, i32 0, i32 0, null, i32 0, i1 false, i32 ()* @main, null, null, metadata !2, i32 3} ; [ DW_TAG_subprogram ] [line 3] [def] [main]
!5 = metadata !{i32 786473, metadata !1}          ; [ DW_TAG_file_type ] [/home/lhellebr/bc/new/SPA/SPA/examples/test1A.c]
!6 = metadata !{i32 786453, i32 0, null, metadata !"", i32 0, i64 0, i64 0, i64 0, i32 0, null, metadata !7, i32 0, null, null, null} ; [ DW_TAG_subroutine_type ] [line 0, size 0, align 0, offset 0] [from ]
!7 = metadata !{metadata !8}
!8 = metadata !{i32 786468, null, null, metadata !"int", i32 0, i64 32, i64 32, i64 0, i32 0, i32 5} ; [ DW_TAG_base_type ] [int] [line 0, size 32, align 32, offset 0, enc DW_ATE_signed]
!9 = metadata !{i32 2, metadata !"Dwarf Version", i32 4}
!10 = metadata !{i32 1, metadata !"Debug Info Version", i32 1}
!11 = metadata !{metadata !"clang version 3.5.0 (trunk 205669)"}
!12 = metadata !{i32 786688, metadata !4, metadata !"i", metadata !5, i32 4, metadata !8, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [i] [line 4]
!13 = metadata !{i32 4, i32 7, metadata !4, null}
!14 = metadata !{i32 4, i32 3, metadata !4, null}
!15 = metadata !{i32 786688, metadata !4, metadata !"j", metadata !5, i32 5, metadata !16, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [j] [line 5]
!16 = metadata !{i32 786447, null, null, metadata !"", i32 0, i64 64, i64 64, i64 0, i32 0, metadata !8} ; [ DW_TAG_pointer_type ] [line 0, size 64, align 64, offset 0] [from int]
!17 = metadata !{i32 5, i32 8, metadata !4, null}
!18 = metadata !{i32 5, i32 3, metadata !4, null}
!19 = metadata !{i32 6, i32 3, metadata !4, null}
!20 = metadata !{i32 7, i32 3, metadata !4, null}
```

Figure 4.10: False positive example

```
int main(void){
  int i = 42, j = 42;
  int *k = &i;
  k = &j;
  i = (*k)++;
  return i;
}
```

Chapter 5

Implementation and testing

In this chapter, we will briefly describe the implementation and show tests and their results.

5.1 Implementation

The project consists of the Clang plugin part (`SPA.cpp`, `LvalueTable.cpp`), alias analysis part (`run.sh` and `debugrun.sh` bash scripts), and support files like `Makefile`, documentation and examples.

The Clang plugin part consists of two `.cpp` files. For implementation reasons, there are no header files. The `SPA.cpp` file consists of the part connecting the SPA with Clang and of the part traversing the AST and deciding what node should be tagged. The `LvalueTable.cpp` file provides functions for actual tagging of nodes and generating the final constraints.

The alias analysis part is a Bash script `run.sh` that uses the output of the first part, and together with LLVM IR, alias analysis output and debugging information checks the constraints and provides the final output to the user. The `run.sh` script is what user invokes directly. There is also a `debugrun.sh` file automatically generated from a `run.sh` file during a `make` command, additionally showing the constraints, LLVM IR, debugging information, alias analysis output and the final output rather than only the final output. See the example debug output in the figure 5.1.

The Makefile is written so that it automatically downloads latest versions of all the necessary software, such as Clang or LLVM, and installs them together with SPA in the current directory.

The documentation source files in latex are in a `SPA/doc` folder, the installation instructions are in `INSTALL.txt` and very basic info about the SPA is in `README.md`.

5.1.1 Further development

Here we discuss the SPA's potential for further development and possible usability for upstream.

The first part is a Clang plugin. As any other Clang plugin, it uses an interface that is *not considered stable* [2] and would need to be consistently developed should it be a part of upstream. The plugin itself works well and may serve as a starting point for further development.

The second part is not as rigid as the first one. It can be easily replaced by some other script using some other alias analysis implementation. It has a flaw described in 4.3.5. It

Figure 5.1: Full debug output

```
$ ./debugrun.sh SPA/examples/test1A.c
LLVM IR:
define i32 @main() #0 {
  call void @llvm.dbg.declare(metadata !{i32* %i}, metadata !12), !dbg !13
  store i32 0, i32* %i, align 4, !dbg !14
  call void @llvm.dbg.declare(metadata !{i32** %j}, metadata !15), !dbg !17
  store i32* %i, i32** %j, align 8, !dbg !18
  %0 = load i32** %j, align 8, !dbg !19
  %1 = load i32* %0, align 4, !dbg !19
  %inc = add nsw i32 %1, 1, !dbg !19
  store i32 %inc, i32* %0, align 4, !dbg !19
  store i32 %1, i32* %i, align 4, !dbg !19
  %2 = load i32* %i, align 4, !dbg !20
  ret i32 %2, !dbg !20
!0 = metadata !{i32 786449, metadata !1, i32 12, metadata !"clang version 3.5.0 (
    trunk 205669)", i1 false, metadata !"", i32 0, metadata !2, metadata !2,
    metadata !3, metadata !2, metadata !2, metadata !"", i32 1} ; [
    DW_TAG_compile_unit ] [/home/lhellebr/bc/new/SPA/SPA/examples/test1A.c] [
    DW_LANG_C99]
!1 = metadata !{metadata !"SPA/examples/test1A.c", metadata !"/home/lhellebr/bc/new/
    SPA"}
!2 = metadata !{}
!3 = metadata !{metadata !4}
!4 = metadata !{i32 786478, metadata !1, metadata !5, metadata !"main", metadata !"
    main", metadata !"", i32 3, metadata !6, i1 false, i1 true, i32 0, i32 0, null,
    i32 0, i1 false, i32 ()* @main, null, null, metadata !2, i32 3} ; [
    DW_TAG_subprogram ] [line 3] [def] [main]
!5 = metadata !{i32 786473, metadata !1}          ; [ DW_TAG_file_type ] [/home/
    lhellebr/bc/new/SPA/SPA/examples/test1A.c]
!6 = metadata !{i32 786453, i32 0, null, metadata !"", i32 0, i64 0, i64 0, i64 0,
    i32 0, null, metadata !7, i32 0, null, null, null} ; [ DW_TAG_subroutine_type ]
    [line 0, size 0, align 0, offset 0] [from ]
!7 = metadata !{metadata !8}
!8 = metadata !{i32 786468, null, null, metadata !"int", i32 0, i64 32, i64 32, i64
    0, i32 0, i32 5} ; [ DW_TAG_base_type ] [int] [line 0, size 32, align 32, offset
     0, enc DW_ATE_signed]
!9 = metadata !{i32 2, metadata !"Dwarf Version", i32 4}
!10 = metadata !{i32 1, metadata !"Debug Info Version", i32 1}
!11 = metadata !{metadata !"clang version 3.5.0 (trunk 205669)"}
!12 = metadata !{i32 786688, metadata !4, metadata !"i", metadata !5, i32 4,
    metadata !8, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [i] [line 4]
!13 = metadata !{i32 4, i32 7, metadata !4, null}
!14 = metadata !{i32 4, i32 3, metadata !4, null}
!15 = metadata !{i32 786688, metadata !4, metadata !"j", metadata !5, i32 5,
    metadata !16, i32 0, i32 0} ; [ DW_TAG_auto_variable ] [j] [line 5]
!16 = metadata !{i32 786447, null, null, metadata !"", i32 0, i64 64, i64 64, i64 0,
     i32 0, metadata !8} ; [ DW_TAG_pointer_type ] [line 0, size 64, align 64,
    offset 0] [from int]
!17 = metadata !{i32 5, i32 8, metadata !4, null}
!18 = metadata !{i32 5, i32 3, metadata !4, null}
!19 = metadata !{i32 6, i32 3, metadata !4, null}
!20 = metadata !{i32 7, i32 3, metadata !4, null}

Aliases:
main 0 i

Constraints:
main 6 3 6 3 *j i

Translated aliases:
main 6 3 any any *j i

Results:
Possible undefined behavior in function "main" at [6,3] - "*j" aliases with "i"
```

relies on an exact output format of alias analysis which is, however, not considered stable. This part is innovative in means of mapping the original variable name and location to a temporary variable via debugging information, but should not be considered stable nor should any further development be based on it.

As to including the whole SPA to upstream, there is generally one issue that makes it difficult and arguably useless: it doesn't find the final results during the compilation. It merely generates constraints which need to be further checked, that is what `run.sh` does. That means this check can't be transparently added to the Clang compile-time checks without major changes in the Clang's architecture.

Potential further development, other than debugging, is thus making the whole tool invokable just as a Clang plugin.

5.2 Testing

We have tested the application using a prepared set of test cases. These can also be effectively used as demonstration of the undefined behavior problem and the principles of the SPA. In this section, we will describe test cases and statistically analyze the results.

5.2.1 Test cases

Currently there are 24 tests. Some of them use pointers. Results of some of them are expected to be positive (undefined behavior found), some are expected to be negative. See details in the table:

Figure 5.2: Test types

	Positive	Negative
With pointers	7	6
Without pointers	9	2

The tests vary in constructions used, use different data types, statement types and undefined behavior origins. Most of them are in a pointer and non-pointer version. They are part of both the repository and the attached source codes.

Some of these cases are: undefined behavior while indexing arrays, functions, ignoring compound statements, correct resolving of sequence points in a ternary operator, tricks to make alias analysis fail, etc. Tests were designed with the design of the tool and its flaws in mind so they can be considered as worst-case scenarios.

5.2.2 Test results

The results are shown in the Figure 5.3

As we can see, there were no false negatives, which means that the SPA never "missed" an actual undefined behavior. In this test set, we have found two false positives.

One of these false positives was caused by the second phase and the flaw described in the second paragraph of 4.3.5. If two variables alias at some point in the function, they are considered as aliasing to the end of the same function even if they do not alias anymore. This was necessary due to insufficient information we can get from the alias analysis output. In it solely a matter of the second phase implementation and if we decide to use some other

Figure 5.3: Test results

True Positives	16
True Negatives	6
False Positives	2
False Negatives	0
Sensitivity	**100%**
Specificity	**75%**
Fall-out	25%
Positive predictive value	**89%**
Negative predictive value	**100%**
False discovery rate	11%
Accuracy	**92%**

alias analysis, this problem may not occur at all. The test was created with this flaw specifically in mind.

The second false positive was caused by our approach to the function side effects on their parameters. In 4.1.1, we described it as "if the function *can* use its argument as an L-value (meaning it has the object pointer or some object it can get it from as an argument), we will assume it *does*". This can not be considered a bug in the implementation. It is the flaw we expected this approach to have. In the referenced chapter, we argue why this is sufficient and changing this approach would render the SPA harder to reuse and inconsistent with the philosophy of the tool.

5.2.3 Test results evaluation

After some statistical analysis, we see that, based on this test set, if there is no undefined behavior detected, there is certainly no undefined behavior (related to sequence points and side effects). If there is some undefined behavior detected, there is a 89% chance there actually is undefined behavior.

The overall accuracy of the SPA is 91% and the only error in the set is caused by the second phase of the SPA.

The run took about 0.04 s on the testing machine (Intel Core i7 @ 2900 Mhz) on Example 2. On small files, it is generally fast enough to be used as a part of compilation.

The tool has also been run on longer source codes such as school projects. The Constraint Generating part runs very fast on programs of almost any size – it only takes a few milliseconds. The Constraint Checking part can take very long even on medium-sized files – on a relatively small file with 203 constraints generated within 3 functions, it takes several minutes to check them. This is probably due to implementation in Bash and the fact that the output of the alias analysis is not structured very well and needs to be parsed.

Chapter 6

Conclusion

We have implemented a tool that searches for undefined behavior related to side effects and sequence points in the C language programs. The tool, called Sequence Point Analyzer, consists of two parts. The first part generates a list of *possible* undefined behavior occurrences. The second part verifies these constraints and generates the final user-readable output. The first part is implemented as a Clang plugin, the second as a Bash script using alias analysis in LLVM.

During designing the tool, we had to solve multiple problems. The Constraint Generating part needed an algorithm for Abstract Syntax Tree traversal. This has been achieved by starting at each DeclRefExpr node and traversing to the root. DeclRefExpr is always a leaf of the AST. We also stated that if a node implies a side effect on certain variable, all its parent nodes have side effect on that same variable, too.

The Constraint Checking part had to solve problems of mapping the LLVM IR to the original source code. This was necessary because a construct leading to undefined behavior in C is well-defined after translation to the LLVM IR. We therefore need to get the original names of temporary variables and the original location of the statement with undefined behavior. This was achieved by parsing certain parts of the LLVM IR and using the debugging information.

The SPA has been tested and the tests have shown the accuracy of 91%, positive predictive value of 89% and negative predictive value of 100%. This means, based on the tests, that if the program has undefined behavior related to sequence points, it will be detected. Also, if an undefined behavior is detected, there is a 89% chance it is not a false positive and the detected construct is actual undefined behavior. The only failed tests failed due to the known flaws originating in alias analysis giving not enough information.

The tool is currently probably not useful in the upstream due to its two-phase architecture which doesn't allow for transparent compile-time checks in Clang. It can be, however, used as a starting point for further development, particularly its first part which seems to work well.

Bibliography

[1] Astrée Run-Time Error Analyzer. <http://www.absint.com/astree/index.htm>. [Online; Accessed: 2014-05-12].

[2] Choosing the Right Interface for Your Application. <http://clang.llvm.org/docs/Tooling.html>. [Online; Accessed: 2014-05-07].

[3] clang: a C language family frontend for LLVM. <http://clang.llvm.org>. [Online; Accessed: 2014-01-29].

[4] Clang vs Other Open Source Compilers. <http://clang.llvm.org/comparison.html>. [Online; Accessed: 2014-01-22].

[5] CompCert. <http://compcert.inria.fr/>. [Online; Accessed: 2014-01-29].

[6] Coverity. <http://www.coverity.com/>. [Online; Accessed: 2014-05-12].

[7] Cppcheck. <http://cppcheck.sourceforge.net/>. [Online; Accessed: 2014-05-12].

[8] GCC, the GNU Compiler Collection. <http://gcc.gnu.org>. [Online; Accessed: 2014-01-29].

[9] Getting Started: Building and Running Clang. <http://clang.llvm.org/get_started.html>. [Online; Accessed: 2014-05-07].

[10] PVS-Studio. <http://www.viva64.com/en/d/0162/>. [Online; Accessed: 2014-01-30].

[11] What Is Static Analysis? And Why Is It Important To Software Testing. <http://www.codeexcellence.com/2012/05/what-is-static-analysis-and-why-is-it-important-to-software-testing/>. [Online; Accessed: 2014-05-13].

[12] Chucky Ellison and Grigore Rosu. c-semantics. <http://code.google.com/p/c-semantics/>. [Online; Accessed: 2014-01-29].

[13] ISO. Information technology – Programming languages – C. ISO 9899:1011, International Organization for Standardization, Geneva, Switzerland, 2011.

[14] Robbert Krebbers. Non-determinism and sequence points in C. <http://gallium.inria.fr/blog/non-determinism-and-sequence-points-in-c/>. [Online; Accessed: 2014-01-29].

[15] Thomas Lenherr. Taxonomy and applications of alias analysis. Master's thesis, Eidgenössische Technische Hochschule Zürich, Institute of Computer Systems, Zürich, September 25 2008. Adviser: Dr. Thomas Wahl.

[16] John Regehr. What Other Dynamic Checkers for C/C++ are Needed? <http://blog.regehr.org/archives/966>. [Online; Accessed: 2014-01-30].

[17] Duncan Sands. A very basic doubt about LLVM Alias Analysis. <http://lists.cs.uiuc.edu/pipermail/llvmdev/2010-February/029411.html>. [Online; Accessed: 2014-05-06].

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[19] WG14. N926 – Sequence Point Analysis. <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n926.htm>. [Online; Accessed: 2014-01-30].

[20] WG14. N1570 – Information technology – Programming languages – C – Committee draft. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf>, 2011. [Online; accessed 2014-01-04].

Appendix A

DVD contents

The DVD contains exactly the same content as the git repository at the time of burning the DVD. The only difference is that it also contains this documentation in pdf. The directory structure is as follows:

- Makefile – the only thing you should need to make the whole tool running
- README.md and INSTALL.txt – basic information about the tool and the installation instructions
- SPA – the source files
  - doc – the source files for generating this documentation
  - examples – the source files of tests/examples
    - negative – the source files of tests that are not supposed to find any undefined behavior
  - SPA.cpp, LvalueTable.cpp, run.sh – the source code of the tool itself, as described in the "Implementation and testing" chapter
- projekt.pdf – this documentation

Appendix B

Installation and usage

In this chapter, we will discuss the software requirements of the SPA, its automated installation and manual installation. We will discuss the usage from user's point of view.

B.1 Requirements

The tool should run on any system capable of compiling and running LLVM and Clang *and* invoking a Bash script in POSIX environment. It has been tested on Linux, specifically multiple versions of Fedora 17 and 19, both x86_64 architecture.

Clang is only required for the first part and full LLVM for the second part. The SPA uses the interfaces of Clang and LLVM that are both considered unstable so it might require specific versions of these tools. It has been tested on Clang 3.5.0 and LLVM 3.5.0.

The best way to download the SPA is Git. The Makefile itself uses SVN which is therefore required for the automated installation.

B.2 Installation

1. If you do not have the SPA, you need to download it, for example by

   ```
   git clone https://github.com/KamikazeCZ/SPA.git
   ```

2. Go to the newly created repository directory

3. For automated installation, run

   ```
   make
   ```

B.3 Manual installation

For manual installation, you need to download Clang and LLVM. Refer to points 1 to 5 of the Get Started guide [9]. You do not need Clang extra tools.

After following these instructions, you should have a `llvm` directory with the source of both LLVM and Clang. Copy the SPA folder (the one *inside* the repository) to `llvm/tools/clang/tools`. Also copy the `SPA/top-level-makefile/Makefile` file to `llvm/tools/clang/tools`.

This should have added the SPA as a Clang plugin and you can continue by point 6 of the Get Started guide.

B.4 Usage

To verify the correct installation, run

```
make run
```

This is equivalent to running

```
./run SPA/examples/test1A.c
```

Running `./run <file.c>` is a standard way to run the whole SPA on certain file. It is a script executing both parts of the SPA. To only run the first part, run

```
build/Release+Asserts/bin/clang <file.c>
```

For debug mode, use

```
./debugrun <file.c>
```

instead.

To generate this documentation, run `make` in `SPA/doc`.